# Goods Transportation with Family-Split Penalties. An heuristic inspired by GRASP

**Alberto Bolgiani** *a.bolgiani@studenti.unibs.it*

**Giorgio Bonardi** *g.bonardi016@studenti.unibs.it*

**Raul Iosif Cocis** *r.cocis@studenti.unibs.it*

**Youssef El Fadi** *y.elfadi@studenti.unibs.it*

## 1 Model Improvements

The ILP formulation that has been provided is the following:

$$\max \quad \sum_{j \in F} (p_f x_j - \delta_j s_h), \tag{1}$$

$$\sum_{k \in K} y_{ik} = x_j, \qquad \forall j \in F, \forall i \in F_j, \tag{2}$$

$$\sum_{i \in I} w_{ir} y_{ik} \leq C_{kr}, \qquad \forall k \in K, \forall r \in R, \tag{3}$$

$$\sum_{i \in F_j} y_{ik} \leq |F_j| z_{jk}, \qquad \forall j \in F, \forall k \in K, \tag{4}$$

$$\sum_{k \in K} z_{jk} - 1 \leq s_j, \qquad \forall j \in F, \tag{5}$$

$$x_j,\, y_{jk},\, z_{jk} \in \{0, 1\}, \qquad \forall i \in I, \forall j \in F, \forall k \in K, \tag{6}$$

$$s_j \in \mathbb{N}, \qquad \forall j \in F. \tag{7}$$

We have decided to implement some improvements to the given ILP formulation.

### 1.1 Modified constraints

We have modified the fourth pre-existing set of constraints as follows:

$$\sum_{i \in F_j} y_{ik} \leq n_{maxItems} * z_{jk}, \qquad \forall j \in F, \forall k \in K \tag{4}$$

In which $n_{maxItems}$ represents the maximum number of items $i \in F_j$ that can be inserted inside the knapsack $k \in K$. In order to calculate the value of the variable $n_{maxItems}$, for every resource $r \in R$ we order by resource consumption all the items belonging to the family and we compute the number of items according to the specific resource that can fit inside the knapsack. Once we have computed those values, $n_{maxItems}$ is set to the minimum of those values.

### 1.2 Added constraints

We have also added a new set of constraints:

$$s_j \leq |F_j| - 1 \qquad \forall j \in F \qquad (8)$$

This type of constraint sets an upper bound on the number of splits that a single family $j$ can have. The upper bound value is equal to the cardinality of the family minus 1. This is because the number of splits will always be less by at least one than the number of items that belong to the same family.

### 1.3 Results with respect to the original ILP formulation

We confronted the original ILP formulation with the one introducing our changes by performing two different runs of Gurobi with a time limit of 3600 seconds each.

| Gurobi - Original Model | | |
|---|---|---|
| Instance | Time(s) | Obj. Function |
| 1 | 3600 | 93856 |
| 2 | 3600 | 277616 |
| 3 | 3600 | 732887 |
| 4 | 3600 | 84389 |
| 5 | 3600 | 266990 |
| 6 | 3600 | 669251 |
| 7 | 3600 | 235073 |
| 8 | 3600 | 693746 |
| 9 | 3600 | 1440176 |
| 10 | 3600 | 495760 |
| 11 | 3600 | 1301603 |
| 12 | 3600 | 3005515 |

| Gurobi - New Model | | | |
|---|---|---|---|
| Instance | Time(s) | Obj. Function | Gap % |
| 1 | 3600 | 93856 | 0,00% |
| 2 | 3600 | 280530 | 1,05% |
| 3 | 3600 | 736122 | 0,44% |
| 4 | 3600 | 84389 | 0,00% |
| 5 | 3600 | 266008 | -0,37% |
| 6 | 3600 | 687465 | 2,72% |
| 7 | 3600 | 235073 | 0,00% |
| 8 | 3600 | 691542 | -0,32% |
| 9 | 3600 | 1537899 | 6,79% |
| 10 | 3600 | 496196 | 0,09% |
| 11 | 3600 | 1288404 | -1,01% |
| 12 | 3600 | 2961025 | -1,48% |

As seen in the tables, there are five instances in which the new ILP formulation performes better than the original one, with the maximum improvement amounting to 6,79%. There are only four instances in which the new ILP formulation performes worse than the original one. In any case, the new values found were only worse by less than 1,5%.

## 2 Algorithm description

### 2.1 General idea

**Algorithm 1** General Algorithm

```
1: function GRASP(instance)
2:     bestSolution ← null
3:     bestObjValue ← null
4:
5:     while (time limit not reached or stale time not reached) do
6:         solutionCP ← constructivePhase(instance)
7:         newObjValue ← calculateObjValue(solutionCP)
8:
9:         if newObjValue > bestObjValue then
10:            bestObjValue ← newObjValue
11:            bestSolution ← solutionCP
12:            resetStaleTime()
13:        end if
14:    end while
15:
16:    totalTimeLimitGurobi ← calculateGurobiTime()
17:    familiesSplits ← calculateFamiliesSplits(bestSolution)
18:    solutionGurobi ← gurobiSearch(istance, bestSolution, totalTimeLimitGurobi, familiesSplits)
19:
20:    return solutionGurobi
21: end function
```

The general idea of our algorithm starts with our version of the Constructive Phase of a feasible solution, to which we assign 200 seconds. If a certain amount of time has passed without improving the solution of the Construnctive Phase[1], the Constructive Phase gets interrupted and we calculate the remaining time [2]. This time gets transformed in additional time to be given to Gurobi, in addition to the default time of 600 seconds. Gurobi is executed by starting from the solution found in the first phase and by fixing some of the variables.

### 2.2 Constructive phase

We cycle through the available families, which are all the ones that have yet to be inserted into any knapsack. Then the algorithm creates a Restricted Candidate List (RCL) from which a single family is randomly selected. The RCL contains a percentage of families ordered according to a certain sorting function. The percentage is determined by the variable $\beta$ that is chosen randomly from a pool of values.

[1] which will be referred as "Stale Time"

[2] calculated by substracting the elapsed time from the initial 200 seconds

We tried to utilize different sorting functions, such as:

- profit oriented (ascending)
- penalty oriented (descending)
- $\frac{profit}{penalty}$
- $specialGain$: $\frac{profit-[(familyCardinality-1)*penalty]}{2}$

Once we selected the family to be inserted, we try to insert the whole family in a single knapsack in order to not split it. If a knapsack that can contain the whole family exists then the family is assigned to that specific knapsack. If this happens, then the family is removed from the list containing the available families and the algorithm proceeds to the next iteration. On the contrary, if the family's resource consumption exceeds the capacities of all the knapsacks (i.e. the whole family doesn't fit completely in any knapsack) then the algorithm finds the so called **most problematic item**. This item $I_{max}$ is the one that consumes the most amount of a certain resource $R_{max}$ and the algorithm first looks into the list of knapsacks that have already been used by the family and tries to find one that can fit that item such that the remaining capacity of the knapsack regarding the resource $R_{max}$ is as low as possible once the item has been inserted. If no knapsack is found, the algorithm looks for one outside the list of the already used knapsacks.

$R_{max}$ represents the resource $r$ related to the most problematic item $I_{max}$ that is consumed the most.

We designed two different alternatives to find the most problematic item as follows:

- **Alternative 1**: The most problematic item is the one in which the value $\frac{items[i][r]}{N_{jr}}$ is greater, where: $items[i][r]$ is the capacity required by the item $i$ regarding a certain resource $r$ while $N_{jr}$ represents the sum of the resources consumed by all the items belonging to family $j$ regarding resource $r$, with item $i \in$ family $j$.
- **Alternative 2**: We use the same ratio as before while changing the denominator. For each resource $r$ and for each item $i$ we cycle through all the knapsacks $k$ while calculating the ratio between the quantity of $r$ used by item $i$ and the remaining capacity of knapsack $k$ with respect to resource $r$. The ratio with the maximum value will determine the most problematic item.

Once the most problematic item has been inserted, the algorithm continues by fitting the rest of the family. If a knapsack that can fit the whole family has not been found, then the algorithm recursively searches for the most problematic item and fits it while separating it from the rest of the family. This cycle repeats until the whole family can be fitted inside a knapsack.

If the remaining list of items of the family to be inserted is made up of only one item and no knapsack that can fit it has been found then the algorithm goes back to the previous solution and removes the family from the list of available families and proceeds with the next one in the list.

**Algorithm 2** Constructive phase

```
1: function constructivePhase(instance)
2:     availableFamilies ← instance.families
3:     betaRCL ← getRandomBeta()
4:     solution ← initSolution()
5:     sortedAvailableFamilies ← sortFamiliesByRankingFun(instance, availableFamilies);
6:
7:     while (availableFamilies is not empty) do
8:         randomFamily ← getFamilyFromRCL(instance,sortedAvailableFamilies,betaRCL)
9:         necessaryResources ← calculateNecessaryResources(instance, randomFamily)
10:        itemToInsert ← randomFamily.items
11:        previousSolution ← solution
12:
13:        solution ← recursiveFitFamily(instance,itemToInsert,necessaryResources,solution)
14:
15:        if solution is null then
16:            solution ← previousSolution
17:        end if
18:
19:        sortedAvailableFamilies.remove(randomFamily)
20:    end while
21:
22:    return solution
23: end function
```

**Algorithm 3** Fit the family in the knapsacks

```
1: function RECURSIVEFITFAMILY(instance, itemToInsert, necessaryResources, solution)
2:     knapsack ← findKnapsackToFitWholeFamily(instance, necessaryResources)
3:
4:     if knapsack is found then
5:         assignFamilyToKnapsack(knapsack)
6:         updateSolution()
7:         updateResources()
8:         return solution
9:     end if
10:
11:    if itemToInsert.size == 1 then
12:        return null
13:    end if
14:
15:    maxItemIndex, maxResourceIndex = findMostProblematicItem(instance, itemToInsert, necessaryResources)
16:
17:    knapsack ← findBestKnapsackForProblematicItem(instance, maxItemIndex, maxResourceIndex)
18:
19:    if knapsack is not found then
20:        return null
21:    else
22:        updateSolution()
23:        updateItemToInsert()
24:        updateResources()
25:        return recursiveFitFamily(instance, itemToInsert, necessaryResources, solution)
26:    end if
27: end function
```

### 2.3 Gurobi

As described in **Algorithm 1**, Gurobi is used only in the last phase, after some iterations of the Constructive phase used to find a good initial solution: $solutionCP$. Gurobi is called by starting from $solutionCP$ and by fixing certain variables $x_j$ as follows:

- fix to 1 a percentage $BestFamilies$ of the first best families selected by $solutionCP$, sorted in ascending order by the number of splits.
- fix to 0 a percentage $WorstFamilies$ of the first worst families that have **not** been selected by $solutionCP$, sorted in ascending order by $specialGain$. This sorting technique allow us to take into consideration the worst case, that is the one in which a family is completely split. We would prefer to keep the families that give more profit

while in their worst scenario. Because of this, we fix the families that we consider would give the least profit in their worst scenario.

This allows the algorithm to reduce the feasible solution space, thus allowing Gurobi to analyze fewer solutions in the hopes of finding one closer to the optimal solution in little time.

Before attempting to optimize $solutionCP$ with Gurobi, the following set of constraints is added to the ILP formulation:

$$s_j \leq actualSplit_j \qquad \forall j \in F \qquad (9)$$

where $actualSplit_j$ represents the number of times that family $j$ has been split up in $solutionCP$.

This means that Gurobi will have to find a solution in which if selected, family $j$ can't be split more than the number of times it has been split in $solutionCP$. This allows the algorithm to further accentuate constraint (8).

**Algorithm 4** Gurobi Search

```
1: function GUROBISEARCH(instance,graspSolution,graspRemainingTime,familiesSplits)
2:     initialGurobiSolution ← graspSolution
3:     timeLimit ← graspRemainingTime + timeForGurobi
4:     setGurobiTime(timeLimit)
5:
6:     familiesToSet ← getBestFamiliesSelectedBySplit(instance, graspSolution)
7:     familiesToBan ← getWorstFamiliesNotSelecteddBySpecialGain(instance, graspSolution)
8:
9:     setGurobiXvarsToOne(familiesToSet)
10:    setGurobiXvarsToZero(familiesToBan)
11:
12:    for family in familiesSplits do
13:        newConstraint ← setUBForSplitsGurobi(family.splits)
14:        addConstraintToModel(newConstraint)
15:    end for
16:
17:    gurobiSolution ← optimizeModel(initialGurobiSolution)
18:
19:    if gurobiSolution is feasible then
20:        return gurobiSolution
21:    end if
22:
23:    return Exception()
24: end function
```

# 3 Computational Results

## 3.1 Parameter choice

The parameters chosen to perform the tests are the following:

1. Type of **sorting function**:
   - Sorted by Penalties in non increasing order
   - SpecialGain
2. **Stale time (s)**:
   - 10
   - 20
   - 30
3. How to find the most problematic item
   - Alternative 1
   - Alternative 2

Taking into consideration all the parameters and their possible values, we have executed 12 different runs.

## 3.2 Hardware specifications

All the runs were executed on a computer with Apple M2 CPU with 8 physical cores, 8 logical processors, using up to 8 threads.

## 3.3 Results

By observing the different runs, we can notice that there isn't a specific combination of the parameters that yields a better result in all the different instances.

Because of this, we took note of each instance of each run where a better result than Gurobi's solution is obtained. The next step involved selecting the instances where the value of TTO[3] was greater than 800 seconds [4].

[3]Time to Overtake - value to be explained along the results presentation
[4]which equals the execution time of the heuristic

| | Alternative 1 | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Rank by Penalties | | | | | | Rank by Special Gain | | | | | | |
| | 10 s | | 20 s | | 30 s | | 10 s | | 20 s | | 30 s | | |
| Instance | Obj | TTB | Obj | TTB | Obj | TTB | Obj | TTB | Obj | TTB | Obj | TTB | Gurobi Obj |
| 1 | 92479 | 762 | 92196 | 344 | 91878 | 23 | 91700 | 715 | 91510 | 600 | 92602 | 203 | 93856 |
| 2 | 275974 | 485 | 270874 | 162 | 274582 | 560 | 276796 | 631 | 274742 | 679 | 278658 | 571 | 280530 |
| 3 | 744034 | 777 | 731280 | 705 | 746340 | 760 | 738721 | 768 | 738903 | 652 | 737168 | 719 | 736122 |
| 4 | 83484 | 89 | 83959 | 281 | 83838 | 41 | 81733 | 706 | 82684 | 77 | 83485 | 323 | 84389 |
| 5 | 267615 | 648 | 266561 | 776 | 267559 | 643 | 262445 | 787 | 263313 | 736 | 263006 | 634 | 266008 |
| 6 | 715560 | 518 | 714896 | 340 | 713137 | 530 | 718823 | 595 | 716594 | 576 | 718088 | 608 | 687465 |
| 7 | 233578 | 662 | 233030 | 688 | 234403 | 273 | 232442 | 656 | 234054 | 177 | 231919 | 716 | 235073 |
| 8 | 706400 | 303 | 707630 | 693 | 709534 | 638 | 705375 | 556 | 706113 | 402 | 705602 | 525 | 691542 |
| 9 | 1867516 | 686 | 1861066 | 610 | 1860935 | 763 | 1860107 | 522 | 1851275 | 616 | 1853818 | 602 | 1537899 |
| 10 | 488930 | 415 | 492753 | 558 | 489407 | 702 | 485855 | 461 | 485561 | 491 | 489811 | 580 | 496196 |
| 11 | 1453988 | 549 | 1446817 | 754 | 1451098 | 654 | 1450085 | 776 | 1444889 | 701 | 1439627 | 631 | 1288404 |
| 12 | 3672159 | 661 | 3630968 | 630 | 3651738 | 456 | 3584264 | 528 | 3604693 | 592 | 3590382 | 552 | 2961025 |

Table 1: Results using Alternative 1

| | Alternative 2 | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Rank by Penalties | | | | | | Rank by Special Gain | | | | | | |
| | 10 s | | 20 s | | 30 s | | 10 s | | 20 s | | 30 s | | |
| Instance | Obj | TTB | Obj | TTB | Obj | TTB | Obj | TTB | Obj | TTB | Obj | TTB | Gurobi Obj |
| 1 | 91321 | 241 | 92026 | 208 | 92155 | 96 | 90303 | 149 | 90577 | 295 | 90178 | 155 | 93856 |
| 2 | 273884 | 784 | 273099 | 158 | 276093 | 749 | 277100 | 723 | 277716 | 622 | 276608 | 694 | 280530 |
| 3 | 741541 | 723 | 741649 | 595 | 736098 | 730 | 738474 | 776 | 737103 | 762 | 735962 | 564 | 736122 |
| 4 | 84254 | 552 | 84357 | 765 | 84057 | 143 | 82080 | 576 | 83384 | 596 | 83103 | 508 | 84389 |
| 5 | 266979 | 689 | 265247 | 677 | 269800 | 753 | 263208 | 767 | 264997 | 729 | 263476 | 703 | 266008 |
| 6 | 714622 | 554 | 713681 | 675 | 716284 | 555 | 710347 | 552 | 717144 | 774 | 715765 | 574 | 687465 |
| 7 | 232638 | 747 | 234504 | 574 | 233482 | 393 | 232568 | 586 | 231103 | 746 | 231494 | 750 | 235073 |
| 8 | 710081 | 515 | 706760 | 645 | 707273 | 440 | 704957 | 543 | 704423 | 479 | 704477 | 308 | 691542 |
| 9 | 1869990 | 700 | 1860439 | 493 | 1869189 | 723 | 1855986 | 598 | 1853438 | 576 | 1859192 | 559 | 1537899 |
| 10 | 491684 | 668 | 487746 | 622 | 487396 | 587 | 486332 | 506 | 485221 | 647 | 484999 | 678 | 496196 |
| 11 | 1451498 | 777 | 1452891 | 718 | 1455694 | 761 | 1446270 | 744 | 1448493 | 732 | 1441531 | 643 | 1288404 |
| 12 | 3658306 | 673 | 3654461 | 528 | 3642287 | 608 | 3582465 | 651 | 3605813 | 575 | 3602770 | 576 | 2961025 |

Table 2: Results using Alternative 2

Among all the combinations of parameters, there are three that offer a better solution in 8 out of 12 instances, meanwhile the other combinations provide a better solution in 7 instances out of 12.

These combinations are the following:

- Alternative 1, Sorted by Penalties, Stale Time: 30s.
- Alternative 1, Sorted by SpecialGain, Stale Time: 20s.
- Alternative 2, Sorted by Penalties, Stale Time: 20s.

The run that performs the best is the first one.

This specific run has been chosen because it has the most number of instances in which the value of TTO is set to NaN.

The columns' meaning is:

- **Objective CP**: solution found by the constructive phase.
- **Objective Heuristic**: best solution found by our heuristic.
- **Gap H.**: improvement of **Objective Heuristic** with respect to **Objective CP**.
- **Time To Best Heuristic**: the time needed to find the **Objective Heuristic** the first time.
- **Objective Gurobi**: best solution found by Gurobi.
- **Time to Overtake**: the time needed by Gurobi to find a better solution than **Objective Heuristic**. In the case in which Gurobi couldn't find a better solution then TTO is set to NaN.
- **Gap**: difference between **Objective Gurobi** and **Objective Heuristic**. A positive value means that **Objective Heuristic** is better than **Objective Gurobi**.
- **Time To Best Gurobi**: time needed by Gurobi to find **Objective Gurobi**.
- **Families Inserted**: percentage of families that have been selected in the solution

| Instance | Obj CP | Obj Heuristic | Gap H. % | TTB Heuristic(s) | Obj Gurobi | TTO(s) | Gap % | TTB Gurobi(s) | Fam. Inserted % |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 84848 | 91878 | 7.65% | 23 | 93856 | 65 | -2,11% | 515 | 23,88% |
| 2 | 256393 | 274582 | 6.62% | 560 | 280530 | 295 | -2,12% | 3198 | 36,64% |
| 3 | 688525 | 746340 | 7.75% | 760 | 736122 | NaN | 1,39% | 3550 | 65,35% |
| 4 | 78825 | 83838 | 5.98% | 41 | 84389 | 123 | -0,65% | 677 | 22,39% |
| 5 | 249454 | 267559 | 6.77% | 643 | 266008 | NaN | 0,58% | 3299 | 35,60% |
| 6 | 666453 | 713137 | 6,55% | 530 | 687465 | NaN | 3,73% | 3600 | 59,90% |
| 7 | 219931 | 234403 | 6,17% | 273 | 235073 | 837 | -0,29% | 3223 | 23,95% |
| 8 | 663345 | 709534 | 6,51% | 638 | 691542 | NaN | 2,60% | 3550 | 36,36% |
| 9 | 1765943 | 1860935 | 5,10% | 763 | 1537899 | NaN | 21,01% | 3016 | 64,94% |
| 10 | 466381 | 489407 | 4,70% | 702 | 496196 | 415 | -1,37% | 3464 | 25,75% |
| 11 | 1374521 | 1451098 | 5,28% | 654 | 1288404 | NaN | 12,63% | 3407 | 37,61% |
| 12 | 3618115 | 3651738 | 0,92% | 456 | 2961025 | NaN | 23,33% | 1513 | 65,9% |

Table 3: Best run vs Gurobi New ILP Formulation

We can observe that the average value of Gap H. % is approximately 6.00%, which means that the Constructive Phase alone finds a good solution.

We can notice also that in this specific run there are two instances (Instance 1 and 4) in which the value of **TTB Heuristic** is really low. This means that the heuristic spent most of the time without finding a better solution. Because of this we could improve the heuristic by introducing an equivalent to the **Stale Time** regarding the execution of Gurobi of ~150/200s.

By implementing this improvement it becomes possible to launch different runs with different initial solutions.